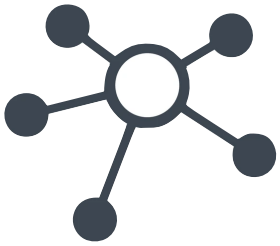
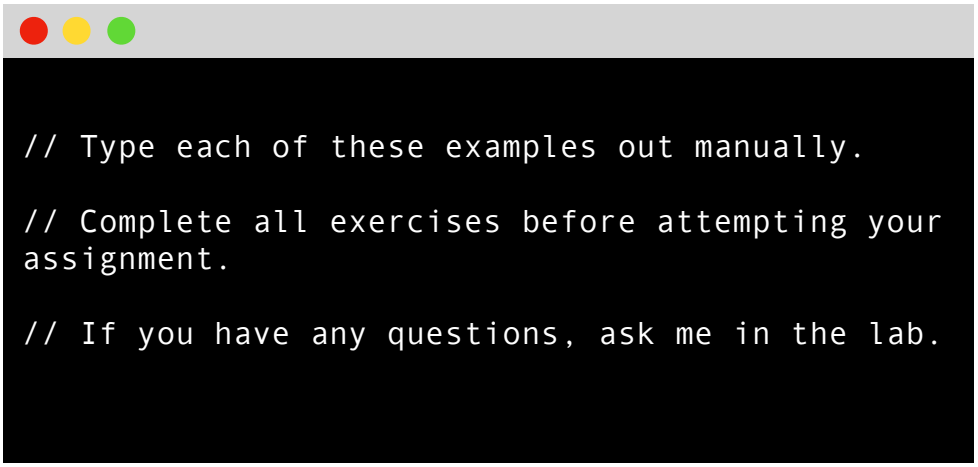
MPI_Notes

Distributed Systems MPI Tutorials



In this set of notes I will show you a series of applications that will use the MPI framework in order to build a distributed system. MPI is far more advanced than Java RMI and is a continually evolving standard. You should have previously installed the necessary MPI libraries for your OS (Windows/Linux/OSX) and completed a Hello World programme.

Follow the provided installation/configuration instructions (on Moodle) for your operating system, and run a practice 'HelloWorld' application before attempting these tutorials.



```
// Type each of these examples out manually.

// Complete all exercises before attempting your
assignment.

// If you have any questions, ask me in the lab.
```

You should type out each of these examples manually. This will help you to practice and serve as muscle memory for when you are building your own MPI programs. Source code files will not be provided, you must type out and complete these tutorials on your own.

Each tutorial highlights a different aspect of the MPI library, so be sure to complete all the examples before starting your assignment. They include explanations of how each part functions and provide guidance throughout the process. Make sure to read this entire document carefully, and if you have questions, ask your lecturer during the lab.

MPI_Notes Tutorials

Click to navigate to the corresponding tutorial

App 01: basic introduction to MPI

In this application we will introduce you to MPI programming. We will start with the 4 most common tasks that will occur in all MPI programs. The code that you see here is C++ based, but as you will see it is similar to Java but with a few minor differences.

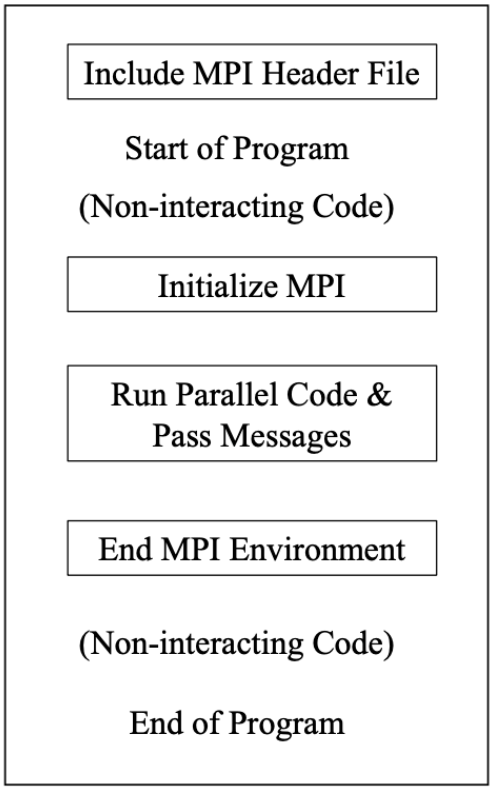
01) create a new source file and give it the following code:
(update your make file with details of the source file if necessary)

```
// MPI_notes - App01 - Basic Introduction to MPI
/** simple program to test the MPI stuff to see if it works **/

/** includes - import necessary libraries and header files**/
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {
    // first thing that we must do at the start of an MPI program
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // one of the important tasks we have to establish is how many processes are
    // in this MPI instance. This tells us who and what we have to communicate with
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    std::cout << "world size is: " << world_size << std::endl;

    // another important job is finding out which rank we are. We can use this rank number
    // to assign separate jobs to different mpi units
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    std::cout << "world rank is: " << world_rank << std::endl;



    // before ending the application,always finalise everything so MPI can shut down properly
    MPI_Finalize();

    // simple output just to verify that everything is working for us
    std::cout << "it works!" << std::endl;

    //return status 0 to the OS
    return 0;
}
```

- For those of you who are used to Java but not C++:
  - `#include` is equivalent to Java's import statement
  - `int main(int argc, char** argv)` is equivalent to Java's `public static void main(String[] args)`
  - `char**` represents a 2 dimensional array of characters that have unknown width and unknown height. C++ arrays do not have a .length property
  - `&world_size` this is to be read as "take the memory address of" the world_size variable, you will learn about pointers in time so don't worry about it for now

For more info on C++ syntax & structure, see Intro to C++ notes on Moodle.

MPI Programs follow this basic structure:



• All MPI programs must start with a single call to `MPI_Init()` we provide null values for both parameters to ask MPI to setup as default.

• All MPI programs must end with a single call to `MPI_Finalize()` to tell the MPI library to shut down and close all connections to other machines.

• Generally in an MPI program you will want to know how many other processors are in the group. this is why we call `MPI_Comm_size(MPI_COMM_WORLD, &world_size)`

- the first argument is a default communication group that includes all MPI processes that are running the same program.

- the second argument is where the number will be stored. this number will indicate how many units are running the same program

• You will also want to know what process rank you are in the group this is why we call `MPI_Comm_rank` with similar arguments to `MPI_Comm_size`

- this provides the key to communication and task division in MPI.
- All MPI tasks for every processor is coded into the same application. To differentiate between different tasks for different processors you simply use if statements for different rank numbers.
- In most of the examples we will use a world size of four but this can be as many processors as you wish.

02) before the call to MPI_Finalize add in the following code, recompile and rerun:

```
// we will try to send a message from rank 0 that is the nodes rank number
// and we will ask the receiving process to print out that number
if(world_rank == 0) {
    MPI_Send(&world_rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Send(&world_rank, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
    MPI_Send(&world_rank, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
}
else {
    int recieved_data = 2000;
    MPI_Recv(&recieved_data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    std::cout << "rank: " << world_rank << " recieved data from rank: " << recieved_data
    << std::endl;
}
```

A few things to note here:

• `MPI_Send` and `MPI_Recv` are the basic blocking send and receive commands in MPI.

• In this example if a process has a world rank of zero it will send out a simple integer message with a value of zero to process ranks 1, 2, 3. These receivers must know that a message is coming.

- If you call `MPI_Send` but you not have an `MPI_Recv` to match it then your program will deadlock and progress will be impossible.

- `MPI_Send` has the following six parameters:
  - the first parameter is the data that you wish to send. In this case we are sending a single integer which is the world rank of process zero. Note that you must have the address of the variable and not the value itself.
  - the second parameter is the count of items to send. If you have more than one item to send at a time you specify the count here.
  - the third parameter is the datatype that is being communicated over the network. There are default basic datatypes but it is possible to define your own datatypes. We will show examples of this later.
  - the fourth parameter is the rank of the process to receive the message.
  - the fifth parameter is the message tag. This gives you a method of distinguishing between different message types.
  - the sixth parameter is the communication group that the message is to be sent to. Because our programs will be small we will use MPI_COMM_WORLD a lot.

- `MPI_Recv` has the same first six parameters as MPI_Send but with minor differences. The fourth parameter is now the process that you are receiving from not sending to.
  - the last parameter indicates the status of the message that has been received. For something as simple as this we will ignore the status information for now.
- In later examples we will show how to do asynchronous communication.

`Expected output when run using four processes (order may vary):`

```
world size is: 4
world size is: 4
world size is: 4
world size is: 4
world rank is: 0
world rank is: 2
world rank is: 3
world rank is: 1
rank: 1 received data from rank: 0
rank: 3 received data from rank: 0
rank: 2 received data from rank: 0
it works!
```

* note: since the processes run in parallel, the order of output is not guaranteed. This means output can appear in a different order each time the program is run.

BACK TO TOP ^ NEXT TUTORIAL >>

App 02: Master/Slave communication in distributed systems

In the previous example you saw a very simple example of how to communicate between processes. In this application we will show how to do some simple master/slave type communication. We will have one master node and three slave nodes. The master node is the coordinator and tells the slaves what to do, while the slave nodes will wait until they receive a message from the master will do that work and will then wait until the master tells them to shut down.

The context for this example is each slave will generate an array of 100 numbers and will calculate the average of them all. When the master receives all three averages it will calculate the overall average of them. This structure is very useful in map-reduce type tasks or any task where you need a single node to coordinate everything else.

01) clear out the code of your previous program and give it this shell code:

```
// MPI_notes - App 02 - Master/Slave convention
/** MPI example that will send a number of items using send and recieve commands **/

/** includes to import libraries **/
#include <iostream>
#include <mpi.h>

/** messages for communicating tasks **/
int COMPUTE_AVERAGE = 1;
int SEND_AVERAGE = 2;
int SHUTDOWN = 3;

/** the world rank and size that will be useful in many functions **/
int size;
int rank;

// ** master method goes here **


// ** slave method goes here **


int main(int argc, char **argv) {
    // always initialise the MPI library at the start of an MPI program
    MPI_Init(NULL, NULL);

    // one of the important tasks we have to establish is how many processes are
    // in this MPI instance. This tells us who and what we have to communicate with
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // another important job is finding out which rank we are. We can use this rank number
    // to assign separate jobs to different mpi units
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);



    // before ending the application,always finalise everything so MPI can shut down properly
    MPI_Finalize();

    //standard C/C++ thing to do, return 0 to the OS
    return 0;
}
```

02) add in the following code before the call to MPI_Finalise() in the main function:

```
// process zero is the master all other processes are slaves
// call the appropriate function depending on which rank we are
if(rank == 0)
    master();
else
    slave();
```

A few things to note here:

- This is the main reason why you have access to the world rank. It is a unique ID for each process in the group. Thus you can assign specific tasks based on this number.
- In this example since we know we will always have a node with rank 0, we generally use that as the rank for the master node and all other nodes will be considered as slaves.

03) add in this function just before the main function:

```
// method that will act as the master for this program
void master(void) {
    // the total average of all the averages from the nodes
    float total_average = 0;
    // an average that we recieve from a node
    float average = 0;

    // ask all three nodes to compute an average
    for(int i = 1; i < size; i++)
        MPI_Send(&COMPUTE_AVERAGE, 1, MPI_INT, i, 0, MPI_COMM_WORLD);

    std::cout << "Master (0): told all slaves to compute" << std::endl;

    // ask all three nodes to send their average to us
    for(int i = 1; i < size; i++) {
        MPI_Send(&SEND_AVERAGE, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        MPI_Recv(&average, 1, MPI_FLOAT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_average += average;
    }

    // take the average of averages and display the result
    std::cout << "Master (0): average result from all slaves is: "
    << total_average / (size - 1) << std::endl;

    // tell all the nodes to shutdown
    for(int i = 1; i < size; i++) {
        MPI_Send(&SHUTDOWN, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
    std::cout << "Master (0): shutting down all slaves" << std::endl;

}
```

A few things to note here:

- This is the function that will be used by the master node, which will be node 0 as per the if statement in the main method.
- In the first for loop this node sends a message to each of the slave nodes to compute the average of 100 random integers by sending a single integer that indicates an action.
- The second for loop requests each node to send the computed average for its generated list of 100 integers.
  - Note how it does a send() and then looks for an immediate receive(). This is the form of communication you must do when using blocking calls. Later on you will be able to split this in two by using asynchronous sends and receives so you can reuse the thread while MPI is busy handling communication.
- The third loop sends the shut down message which will cause all processes to finish executing.
- It is possible to have the master process do some processing and taking part in the task while the slave threads are computing. In this case the master process will become a coordinator.

04) add in this function just below the definition of the master function:

```
// method that will act as the slave for this program
void slave(void) {
    // the message type that we have recieved
    int message_type = 0;
    float average = 0;

    // keep looping until we recieve a shutdown message
    MPI_Recv(&message_type, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    while(message_type != SHUTDOWN) {
        if(message_type == COMPUTE_AVERAGE) {
            std::cout << "Slave" << rank << ": calculating average..." << std::endl;
            // get the average of 100 random numbers
            srand(rank);
            int sum = 0;
            for(int i = 0; i < 100; i++)
                sum += rand() % 10;
            average = sum / 100.f;

            std::cout << "Slave" << rank << ": sum of 100 ints is " << sum << std::endl;
            std::cout << "Slave" << rank << ": average of 100 ints is "
            << average << std::endl;
            std::cout << "Slave" << rank << ": calculated average" << std::endl;
        }
        else if(message_type == SEND_AVERAGE) {
            MPI_Send(&average, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
            std::cout << "Slave" << rank << ": sent average" << std::endl;
        }
        MPI_Recv(&message_type, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    std::cout << "Slave" << rank << ": shutting down" << std::endl;
}
```

A few things to note here:

- Here the slaves are constantly waiting for a new message from the master process until the shutdown message is sent.
- Note that if you are using a while loop you should receive a message first before going any further. (good place to use a do-while loop)
- Every time the slave receives a message it will look at the message type and from that it will take the appropriate action.
- Finally when the shutdown message is received then the loop will terminate and the function will be exited.

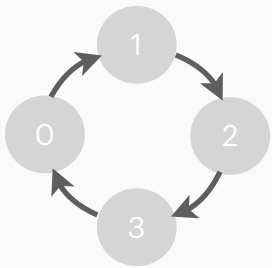
BACK TO TOP ^ NEXT TUTORIAL >>

App 03: building App02 in a ring like structure

It is not necessary to write all algorithms in a master-slave structure. In this example we will replicate the same task using a ring structure of 4 connected processes: 0 → 1 → 2 → 3 → 0. Process 0 will act as the coordinator and will inform Process 1 to start processing. Process 1 will inform Process 2 to start processing and so on.

This method takes advantage of the parallel/concurrent nature of a distributed system a bit better than the previous method. When the results are computed, process 0 will ask process 1 to total the sum by passing its sum to process 1. This continues until process 0 receives the total sum and it will then ask the other processes to shut themselves down. This kind of structure is very useful in matrix multiplication tasks.

01) create a new project with a single source file and add in the following shell code:




```
// MPI_notes - App 03 - App02 in a Ring structure
/** Using a ring structure of 4 connected processes **/

/** includes **/
#include <iostream>
#include <cstdlib>
#include <mpi.h>

/** messages for communicating tasks **/
int COMPUTE_AVERAGE = 1;

/** the world rank and size that will be useful in many functions **/
int world_size;
int world_rank;

// ** coordinator method goes here **


// ** computeAverage method goes here **


int main(int argc, char** argv) {
    // always initialise the MPI library at the start of an MPI program
    MPI_Init(NULL, NULL);

    // one of the important tasks we have to establish is how many processes are
    // in this MPI instance. This tells us who and what we have to communicate with
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // another important job is finding out which rank we are. We can use this rank number
    // to assign separate jobs to different mpi units
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // process zero is the coordinator
    // compute the average of 400 numbers by using ring like communication
    if(world_rank == 0)
        coordinator();
    else
        computeAverage();

    // before ending the application, always finalise everything so MPI can shut down properly
    MPI_Finalize();

    //standard C/C++ thing to do, return control to the OS
    return 0;
}
```

A few things to note here:

- Note that we have removed two of the message types. This is because we will instruct each node to start computation and after that we will follow a strict method for this task.
- We have also set a single node as coordinator and all other nodes are helpers.

02) add the following function above the main function:

```
void coordinator(void) {
    int message;

    // tell the next node that we have to start computing an average
    MPI_Send(&COMPUTE_AVERAGE, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    // we will expect the last node to send us a message to compute our average. get it
    // and ignore it
    MPI_Recv(&message, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // compute our average and print out the results
    srand(world_rank);
    int sum = 0;

    for(int i = 0; i < 100000; i++)
        sum += rand() % 10;
    float average = sum / 100000.f;

    std::cout << "coordinator 0 sum is: " << sum << std::endl; std::cout
    << "coordinator 0 average is: " << average << std::endl;

    // ask node one to compute its average by sending a single floating point value this
    // will add in it's average and pass to node 2 etc
    // when we get the result back we will have the full average
    MPI_Send(&average, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&average, 1, MPI_FLOAT, world_size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // print out the overall average by dividing by the number of nodes and exit
    std::cout << "coordinator 0 total average is: " << average / world_size << std::endl;
}
```

A few things to note here:

- This looks similar to the master function from the previous program. In this case we are also performing computation with this node.
  - First the coordinator must instruct all other nodes in the ring to start computation.
  - It does this by sending a message to node 1.
  - When all nodes have been notified the coordinator receives the compute message from the last node in the ring.
- At this point the coordinator knows that all nodes are processing and therefore can start processing itself.
- When computation is finished the coordinator must get the total average from all nodes. again we use ring like communication here. we send the average onto the next node which will put add its average.
- At the end when all nodes have added the average the coordinator will receive a message from the last node with the fully compiled average

03) add the following function above the main function:

```
void computeAverage(void) {
    int message;
    float current_average;

    // wait till we get the compute message after this tell the next node to start
    MPI_Recv(&message, 1, MPI_INT, (world_rank + world_size - 1) % world_size, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&COMPUTE_AVERAGE, 1, MPI_INT, (world_rank + 1) % world_size, 0, MPI_COMM_WORLD);

    // compute average
    srand(world_rank);
    int sum = 0;
    for(int i = 0; i < 100000; i++)
        sum += rand() % 10;
    float average = sum / 100000.f;

    // print out infomation about our sum and average
    std::cout << "node " << world_rank << " sum is: " << sum << std::endl;
    std::cout << "node " << world_rank << " average is: " << average << std::endl;

    // at some point we will get a message that will contain a current total of the
    // overall average. take that value add our average
    // to it and pass it on to the next node
    MPI_Recv(&current_average, 1, MPI_FLOAT, (world_rank + world_size - 1) % world_size,
             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    current_average += average;

    std::cout << "node " << world_rank << " current average is " << current_average << std::endl;
    MPI_Send(&current_average, 1, MPI_FLOAT, (world_rank + 1) % world_size, 0, MPI_COMM_WORLD);
}
```

A few things to note here:

- Note that the big difference between this method and the slave of the last method is that we have no for loops or if statements. There are no decisions necessary. All coordination is handled by the sending and the receiving of messages.
- At the very start of the function we wait for the compute average message to be received. Once received we immediately pass it onto the next node in the ring and start computing.
- After the average has been computed each node knows that it will then receive a single floating point number that represents a total average from the previous node each node will add its average to this total and pass it onto the next node.
- After this the node will exit and finish.

BACK TO TOP ^ NEXT TUTORIAL >>

App 04: The Hypercube topology 

In this application we will do an addition task similar to the previous three applications. However in this application we will show a hypercube topology. A hypercube is a 4 or more dimensional extension of a cube. Each edge has exactly two nodes regardless of direction. This may seem like a complex structure or topology for a network or communication, however it is efficient at propagating data and messages or for collecting messages or data from the network. A restriction of this network is that there must be a number of nodes that is an exact power of two. It is recommended that you run this with 8 or 16 nodes.

01) start with a fresh .cpp file and add in the following shell code:

```
// MPI_notes - App 04 - The Hypercube topology

/** includes **/
#include <iostream>
#include <cstdlib>
#include <mpi.h>

/** the world rank and size that will be useful in many functions **/
int world_size;
int world_rank;


// ==================== HELPER FUNCTIONS ==========================

// ** hypercubePower method goes here **

// ** initArray method goes here

// ** mostSignificantPower method goes here

// ** computeDirections method goes here

// ** randomSum method goes here


// ============= COORDINATOR =================

// ** coordinator method goes here **


// ================== PARTICIPANT ==================

// ** participant method goes here **



int main(int argc, char** argv) {
    // always initialise the MPI library at the start of an MPI program
    MPI_Init(NULL, NULL);

    // get the total amount of nodes running this programme
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // find out rank of each individual node
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // depending on our rank we may be the coordinator or a participant
    if(world_rank == 0)
        coordinator();
    else
        participant();

    // before ending the application, always finalise everything so MPI can shut down properly
    MPI_Finalize();

    //standard C/C++ thing to do, return control to the OS
    return 0;
}
```

02) add in the following helper methods above the main method:

```
// function to calculate the power of 2 that we are using.
//For hypercubes we need this to figure out how many directions of communication
int hypercubePower(int size) {
    unsigned int power = 1, j = 0;
    for(; power < size; power = power << 1, j++);
    std::cout << j << std::endl;
    return j;
}

// initialises an array with given value and size
void initArray(int *to_init, int value, unsigned int size) {
    for(unsigned int i = 0; i < size; i++)
        to_init[i] = value;
}

// calculates our most significant power
int mostSignficantPower(int rank) {
    int power = 1;
    // to calculate most significant power, keep multiplying by two until we find a
    //number bigger than rank, then go 1 step back

    while(power < rank)
        power <<= 1;
    power >>= 1;

    // return the power when we are finished
    return power;
}

// determines the communication directions based on the given rank
void computeDirections(int *comms_offset, int rank, int hypercube_power) {
    // go through each power: if we have a 0 bit we must add, a 1 bit we must subtract
    for (int i = 1, j = 0; j < hypercube_power; i <<= 1, j++) {
        if (rank & i)
            comms_offset[j] = -i;
        else
            comms_offset[j] = i;
    }
}

// calculates the sum of 100 random integers and returns it
int randomSum(void) {
    int sum = 0;
    for(unsigned int i = 0; i < 100; i++)
        sum += rand() % 10;
    return sum;
}
```

A few things to note here:

- Some of these functions are necessary in a hypercube topology you need to know the exact power of the number of nodes in the topology
- The most significant power is used to determine which directions each node will send messages in and which directions it will receive messages for both distribution and reduction tasks
- Each node can communicate with one and only one node in each direction which will be determined by the individual bits in their rank. if the bit is one the communicating node had that bit set to zero and vice versa.
  - this is why we calculate an offset for each communication direction.

03) Add in the following coordinator method:

```
void coordinator(void) {
    // find out what power is used in the hypercube - tells us how many communication directions we have
    int power = hypercubePower(world_size);
    std::cout << power << std::endl;

    // array that will hold offsets needed to connect with the appropriate nodes in each direction
    int comms_offsets[16];
    initArray(comms_offsets, 0, 16);

    // determine our communication directions based on the bits in our rank
    computeDirections(comms_offsets, world_rank, power);
    for(unsigned int i = 0; i < power; i++)
        std::cout << "rank " << world_rank << " offset " << i << ": " << comms_offsets[i] << std::endl;

    // to start computation, we need to distribute a message for starting the computation
    // in this case a single integer that will represent the dimension that we need to communicate on.
    // this will state which channel we need to send on
    int message = 0;
    for (unsigned int i = 0; i < power; i++) {
        MPI_Send(&i, 1, MPI_INT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD);
        std::cout << "rank " << world_rank << "send message to rank " << world_rank + comms_offsets[i]
        << std::endl;
    }

    // calculate the average of 100 random numbers
    srand(world_rank);
    int sum = randomSum();
    float average = sum / 100.f;
    std::cout << "rank " << world_rank << " average is: " << average << std::endl;

    // send a message to all the other nodes to tell them to sum up their averages
    for (unsigned int i = 0; i < power; i++) {
        MPI_Send(&i, 1, MPI_INT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD);
        std::cout << "rank " << world_rank << "send total message to rank " <<
        world_rank + comms_offsets[i] << std::endl;
    }

    // get the averages from each direction
    float total_average = average;
    for (int message = power - 1; message >= 0; message--) {
        MPI_Recv(&average, 1, MPI_FLOAT, world_rank + comms_offsets[message], 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        std::cout << "rank " << world_rank << " recieved total from " << world_rank + comms_offsets[message]
        << std::endl;
        total_average += average;
    }

    // print out the overall average
    std::cout << "rank " << world_rank << " total and average: " << total_average << " "
    << total_average / world_size << std::endl;
}
```

A few things to note here:

- As you can see from the method above, communicating in a hyper cube even with a simple additional task can be quite complex.
- However the upside of this is that the propagation of a message from originator to all nodes is O(log N) instead of O(N) in a master/slave or ring topology.
- Note that the message we send for coordination is based on the power as this will determine the next communication direction.
- In this example node zero will send a message to nodes 1,2, and 4.

04) Add in the following participant function:

```
void participant(void) {
    // find out what power is used in the hypercube - tells us how many communication directions we have
    int power = hypercubePower(world_size);
    std::cout << power << std::endl;

    // array with offsets that we need to connect with the appropriate nodes in each direction
    int comms_offsets[16];
    initArray(comms_offsets, 0, 16);

    // determine our communication directions based on the bits in our rank
    computeDirections(comms_offsets, world_rank, power);
    for (unsigned int i = 0; i < power; i++)
        std::cout << "rank " << world_rank << " offset " << i << ": " << comms_offsets[i] << std::endl;

    // receive a message to start computation from another node and then propogate it on further
    int message = 0;
    MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    for (message++; message < power; message++) {
        MPI_Send(&message, 1, MPI_INT, world_rank + comms_offsets[message], 0, MPI_COMM_WORLD);
        std::cout << "rank " << world_rank << " send message to rank " <<
        world_rank + comms_offsets[message] << std::endl;
    }
    // calculate the average of 100 random numbers
    srand(world_rank);
    int sum = randomSum();
    float average = sum / 100.f;
    std::cout << "rank " << world_rank << " average is: " << average << std::endl;

    // get a message that will tell us to total our average get this message and retain it
    // send a message to all the other nodes to tell them to sum up their averages
    MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = message + 1; i < power; i++) {
        MPI_Send(&i, 1, MPI_INT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD);
        std::cout << "rank " << world_rank << " sent total message to rank " <<
        world_rank + comms_offsets[i] << std::endl;
    }

    // get the averages from each direction
    float total_average = average;
    for (int i = power - 1; i > message; i--) {
        MPI_Recv(&average, 1, MPI_FLOAT, world_rank + comms_offsets[i], 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        std::cout << "rank " << world_rank << " recieved total from "
        << world_rank + comms_offsets[i] << std::endl;
        total_average += average;
    }
    // print out the overall average
    std::cout << "rank " << world_rank << " total and average: " << total_average << " "
    << total_average / world_size << std::endl;

    // send our average onto the the next node
    std::cout << "rank " << world_rank << " " << comms_offsets[message] << std::endl;
    MPI_Send(&total_average, 1, MPI_FLOAT, world_rank + comms_offsets[message], 0, MPI_COMM_WORLD);
    std::cout << "rank " << world_rank << " sent total average to rank " <<
    world_rank + comms_offsets[message] << std::endl;
}
```

A few things to note here:

- This looks more complex than before but the communication speedup is worth the complexity.
- At first we wait to receive a message from any node, when that message is received it will denote which direction the communication came from. This will determine the amount of sends each node must do:

  - A higher number indicates less sending.
  - A number equal to the power will indicate no sending.
- Computation works as before for the summation of the 100 numbers and then a similar communication message is sent through the network.
- After this the totals must be sent back to the coordinator. This works in the opposite way to sending the communication message. The totals are added and propagated back to the coordinator.
  - This takes advantage of multiple communication links and acts like a reverse flood.

BACK TO TOP ^ **NEXT TUTORIAL >>**

App 05: using MPI's inbuilt broadcasting mechanism

With the previous applications we have shown a need to communicate simple messages using the MPI_Send and MPI_Recv commands. This is such a common operation in MPI applications that there is a special function used for broadcasting a message to all other units in a communicator.

This application shows the most basic example of communicating a message to each and every rank in the communicator. Note that the if statement is not required around the `MPI_Bcast` function. MPI will automatically determine which node is broadcasting and which nodes are receiving.

If you're not communicating with a single node (1-to-1 communication), it's usually better to use MPI_Bcast instead of Send/Recv. Every MPI_Send requires a corresponding MPI_Recv, and having too many of these calls can lead to application crashes if there are even minor errors in them.

01) start with a fresh MPI project and create a single source file

02) add this shell code to your source file:

```
/** MPI_notes - App 05 - Broadcasting using MPI_Bcast **/

/** includes **/
#include <iostream>
#include <mpi.h>


int main(int argc, char** argv) {
    // always initialise the MPI library at the start of an MPI program
    MPI_Init(NULL, NULL);

    // determine the world size
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    //determine our rank in the world
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // print out the rank and size
    std::cout << "rank " << world_rank << " / " << world_size << std::endl;

    // 03 code goes here


    // finalise the MPI library
    MPI_Finalize();

    //return control to the OS
    return 0;
}
```

03) add this code to replace the comment for '03 code goes here':

```
// setup master/slave broadcast
if(world_rank == 0) {
    // broadcast a message to all the other nodes
    int message = 0xDEADBEEF;
    MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD);
    std::cout << "rank 0 broadcasting " << message << std::endl;
}
else {
    // receive a message from the root
    int message = 0;
    MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD);
    std::cout << "rank " << world_rank << " received broadcast of " << message << std::endl;
}
```

A few things to note here:

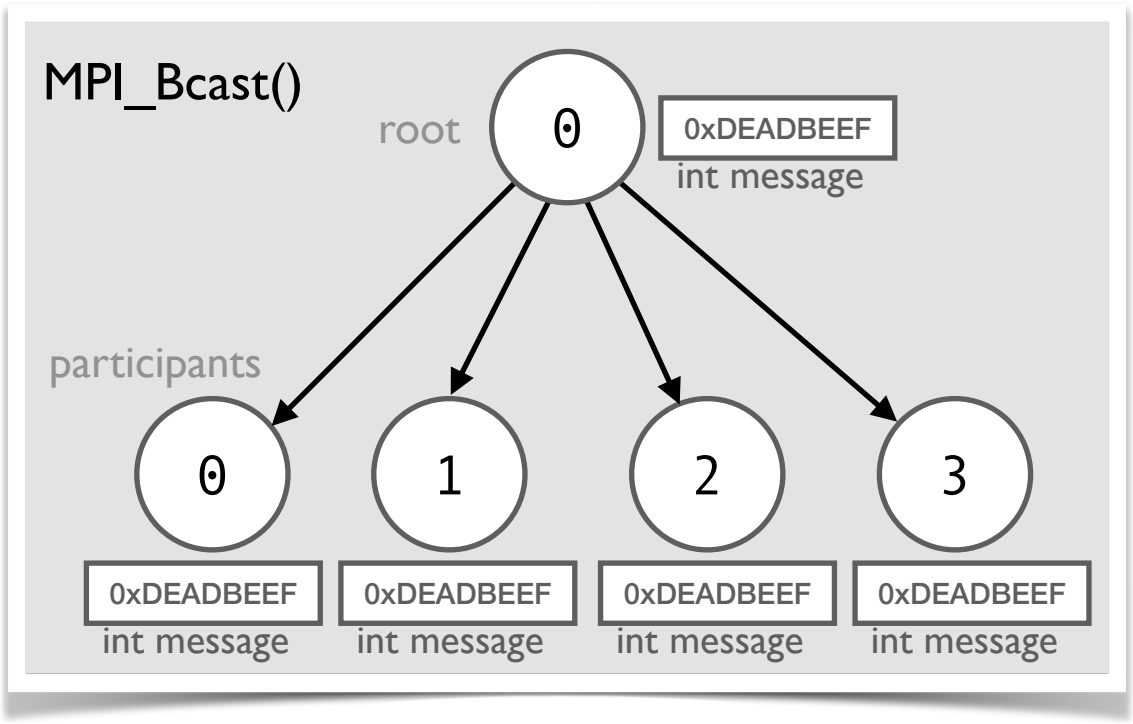
- The if/else statement is not necessary here. The only reason that it is used is to show that `MPI_Bcast` determines who is sending and receiving by itself.

04) now replace the code you added in step 03 with this new code:

```
// setup broadcast without if/else as it is not needed
if(world_rank == 0) {
    // only node 0 has the correct message
    int message = 0xDEADBEEF;

}
else {
    // all other nodes have a blank message variable
    int message = 0;
}
//rank 0 broadcasts the message, thus all nodes now have it
MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD);
std::cout << "rank " << world_rank << "received message: " << message << std::endl;
```

A few things to note here:

- `MPI_Bcast` takes 5 arguments:
  - the first is the buffer to send from or receive into
  - the second is the count of items to send
  - the third is the type of data to send
  - the fourth is the root of this communication. if the root matches the world rank MPI_Bcast will initiate a send. otherwise it will initiate a receive.
  - the final argument is the communicator through which this message will be sent.



BACK TO TOP ^

NEXT TUTORIAL >>

App 06: using MPI_Reduce to total a sum across all nodes

In the examples we have seen so far we have had to communicate our results back to the root node using a combination of sends and receives. This is such a common task within MPI that there is a separate function for doing these reduce operations called `MPI_Reduce`. In this example we show 4 nodes with a single sum communicating to the root node to generate a total sum.

01) start with a fresh project

02) add in this base code to your main source file:

```
/** MPI_notes - App06 - using MPI_Reduce **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    //determine out rank in the world
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // print out the rank and world size
    std::cout << "rank " << world_rank << " out of " << world_size << std::endl;

    //code for 03 goes here


    // finalise the MPI library
    MPI_Finalize();

    // return control to the OS
    return 0;
}
```

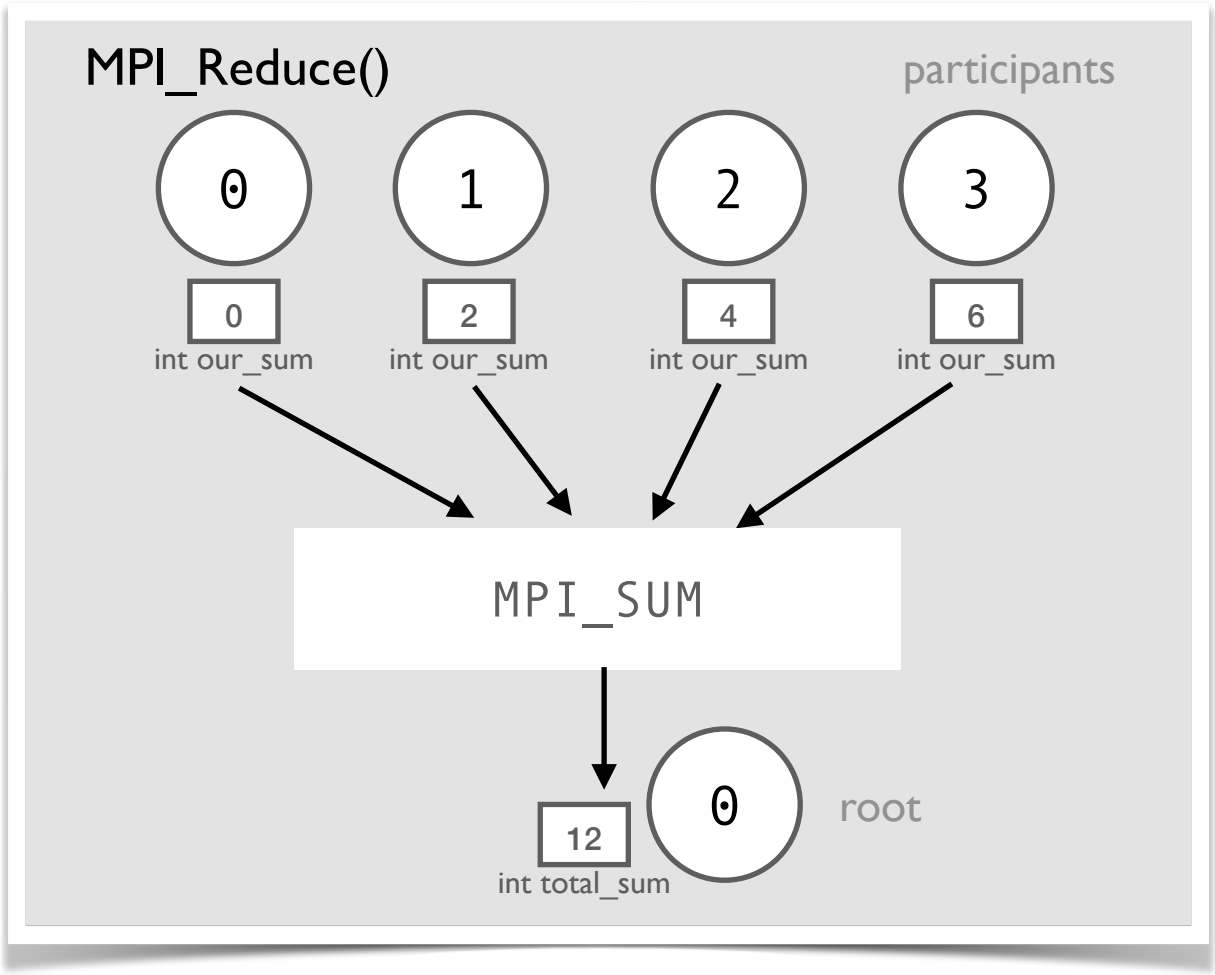
03) add in this code in place of the comment marked "code for 03 goes here":

```
// we have a local sum in each node, we want to reduce that to a
// single value in the root node & display to the user
int our_sum = world_rank * 2;
int total_sum = 0;

MPI_Reduce(&our_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
std::cout << "rank " << world_rank << " - our_sum:" << our_sum << "  total_sum: "
<< total_sum << std::endl;
```

A few things to note here:

- note that there is no decision to be made on who sends or receives the data. `MPI_Reduce` like MPI_Bcast will automatically determine who is the sender and who is the receiver.

- `MPI_Reduce` requires 7 arguments:
  - The first is the local variable that is to be reduced across all nodes. All nodes must implement this variable
  - The second is the destination variable that will contain the reduced value. All nodes must implement this variable but only the root node will store a value in this variable.
  - Third and Fourth are the count and type as before.
  - The fifth argument is the reduce operation that is to be performed on all nodes. MPI_Sum is one such built in operation but there are other built in operations like min, max, bitwise and/or/not, logical and/or/not etc.
  - The sixth argument is the root of the operation. this functions in the same manner as MPI_Bcast.
  - The seventh argument is the communicator on which the operation is performed. As MPI_COMM_WORLD includes all nodes in the program, all nodes participate in the call.



04) try replacing MPI_SUM with MPI_MIN or MPI_MAX to see how the results change:

```
MPI_Reduce(&our_sum, &total_sum, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
```

BACK TO TOP ^ **NEXT TUTORIAL >>**

App 07: using MPI_Barrier to force synchronisation

At certain points during a computation you may require that all nodes finish a certain set of processing tasks first, before they can move onto the next task. In this case you need to force all nodes to synchronise before they can continue. This is achieved by the `MPI_Barrier` function which will wait for all nodes to synchronise before allowing all nodes to continue working.

01) start with a new project
02) in your main source file add in the following base code:

```
/** MPI_notes - App07 - using MPI_Barrier to force synchronisation **/

/** includes **/
#include <iostream>
#include <mpi.h>
#include <cstdlib>
#include <unistd.h>

int main(int argc, char** argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // determine our rank in the world
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // print out the rank and size
    std::cout << "rank " << world_rank << "/" << world_size << std::endl;

    //code for 03 goes here


    // finalise the MPI library
    MPI_Finalize();

    //return control to the OS
    return 0;
}
```

03) add in the following code in place of the comment marked "code for 03 goes here":

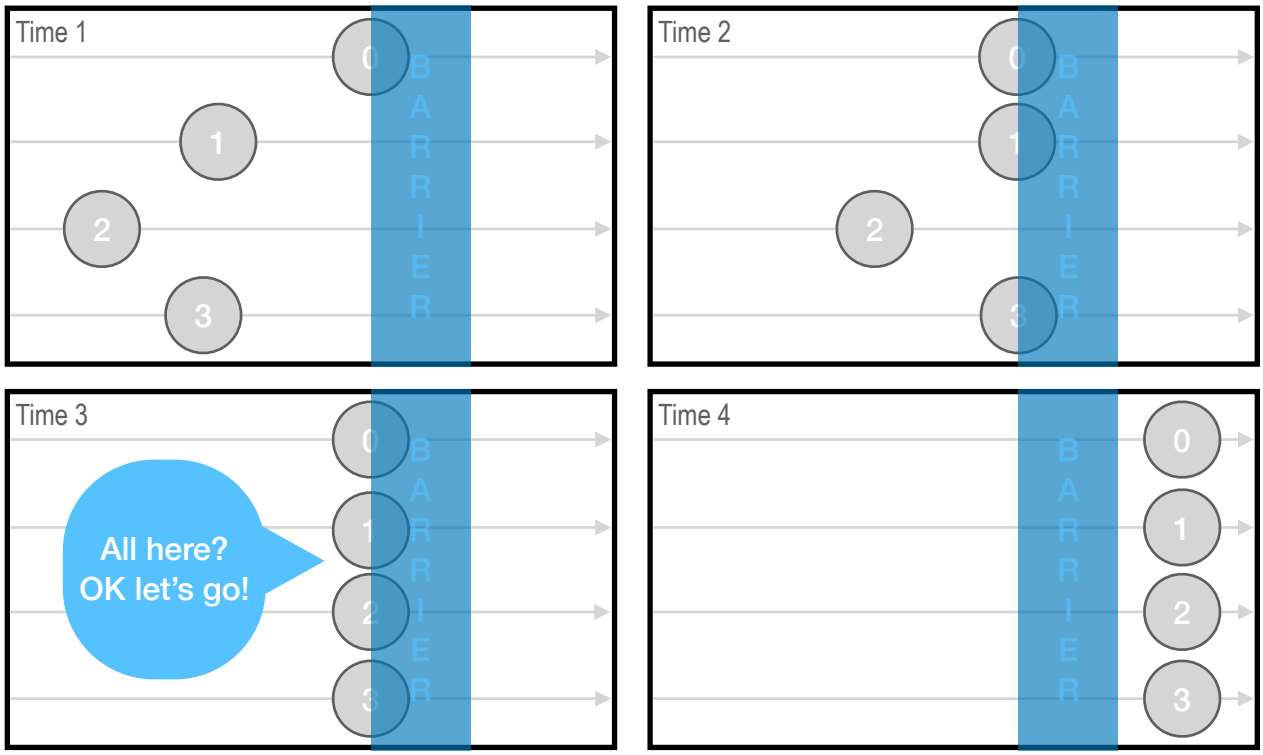
```
// will simulate an operation across many nodes by implementing a random sleep time
// we will get all nodes to synchronise before printing out final message
// ** IF using Unix/Linux systems remove the leading underscore from the call to the sleep command
// (E.g. Mac users use sleep(sleep_time*1000))

srand(world_rank);
int sleep_time = rand() % 5;

std::cout << "node " << world_rank << " sleeping for " << sleep_time << " seconds." << std::endl;
_sleep(sleep_time);
std::cout << "node " << world_rank << " exiting sleep and synchronising" << std::endl;

MPI_Barrier(MPI_COMM_WORLD);
std::cout << "node " << world_rank << " synchronised" << std::endl;
```

Unix users: remove the leading underscore from the sleep command

A few things to note here:

- Here we pick a random sleep time for all nodes between 0 and 5 seconds.
  - This is used to simulate different workloads on each node.
  - when you run this application you will note that all nodes will sleep and synchronise before the final message is displayed

- `MPI_Barrier` forces synchronisation. It accepts a single parameter which is a communicator.
  - The barrier will force all nodes into a cold sleep until all nodes in the communicator have made the MPI_Barrier call.
  - MPI_Barrier will then release all nodes to start computing again.
  - Hence this is why you see the synchronised message at the end of computation.



BACK TO TOP ^

NEXT TUTORIAL >>

App 08: an example of a standard MPI computation application

In previous applications we showed the basics of communication and synchronisation with MPI. However, most MPI applications will send out data from one process to the rest perform some computation and then will reduce those results to a single unit. In this example we show how to distribute data to individual nodes do the computation and collect the result by using the reduce command.

01) start with a fresh MPI project
02) add in the following shell code to your main source file:

```
/** MPI_notes - App08 - a standard MPI Computation **/

/** includes **/
#include <iostream>
#include <mpi.h>


// function that will implement the coordinator job of this application
void coordinator(int world_size) {

}

// function that will implement the participant job of this application
void participant(int world_rank, int world_size) {

}

int main(int argc, char** argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // determine our rank in the world
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // print out the rank and size
    std::cout << "rank " << world_rank << " / " << world_size << std::endl;

    // 0 is the coordinator, the rest are participants of the task
    if(world_rank == 0)
        coordinator(world_size);
    else
        participant(world_rank, world_size);

    // finalise the MPI library
    MPI_Finalize();
}
```

03) add in the following code to the coordinator function:

```
std::cout << "coordinator (rank 0) starting." << std::endl;

// generate 100000 random ints & store them in an array
int values[100000];
for(unsigned int i = 0; i < 100000; i++)
    values[i] = rand() % 10;

// determine the size of each partition by dividing 100000 by world size
// IMPORTANT that the world_size divides this evenly
int partition_size = 100000 / world_size;
std::cout << "coordinator (rank 0) partition size is " << partition_size << std::endl;

// broadcast the partition size to each node so they can setup up memory
MPI_Bcast(&partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
std::cout << "coordinator (rank 0) broadcasted partition size." << std::endl;
```

```
// send out a partition of data to each node
for(unsigned int i = 1; i < world_size; i++) {
    MPI_Send(values + partition_size * i, partition_size, MPI_INT, i, 0, MPI_COMM_WORLD);
    std::cout << "coordinator (rank 0) sent partition to rank " << i << std::endl;
}

// generate an average for our partition
int total = 0;
for(unsigned int i = 0; i < partition_size; i++)
    total += values[i];
float average = (float) total / partition_size;
std::cout << "coordinator (rank 0) average is " << average << std::endl;

// call reduce to get the total average then divide that by the world size
float total_average = 0;
MPI_Reduce(&average, &total_average, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
total_average = total_average / world_size;
std::cout << "total average is " << total_average << std::endl;
```

A few things to note here:

- In this example the root node will generate the data.
  - in this case it is randomly generated but in the majority case it will be stored in files that are to be read and transferred over the network
- The total data is divided to generate equal size partitions that are independent of each other
- The root broadcasts the partition size to all the nodes in the communicator
  - this is to ensure that the other nodes can allocate the necessary memory for the values it will receive.
- In the send command the first argument is `values + partition_size * i`.
  - note that the ampersand (&) is missing. this is because values is an array and not a basic variable. arrays do not require the preceding ampersand.
  - the addition of partition_size * i is to ensure that each node receives the correct data.
- As before, the calculation of the average and the reduction of the values work the same

04) add in the following code to the participant function:

```
std::cout << "participant rank " << world_rank << " starting" << std::endl;

// get partition size from the root and allocate memory as necessary
int partition_size = 0;
MPI_Bcast(&partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
std::cout << "participant rank " << world_rank << " recieved partition size of "
<< partition_size << std::endl;

// allocate memory for our partition
int *partition = new int[partition_size];

// recieve the partition from the root
MPI_Recv(partition, partition_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
std::cout << "participant rank " << world_rank << " recieved partiton from root" << std::endl;

// generate an average for our partition
int total = 0;
for(unsigned int i = 0; i < partition_size; i++)
    total += partition[i];
float average = (float) total / partition_size;
std::cout << "participant rank " << world_rank << " average is " << average << std::endl;

// call reduce to get total average then divide that by the world size
float total_average = 0;
MPI_Reduce(&average, &total_average, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

// finished with memory, so free it
delete[] partition;
```

A few things to note here:

- The first operation is to determine the size of the data that we are receiving. Hence the call to MPI_Bcast.
- After this we can allocate the appropriate array size before receiving the values. Again because partition is an array we omit the & in the MPI_Recv command.
- The rest of the operations for generating the average and reducing the values proceed as before
- Note for those of you coming from Java, in C/C++ we are responsible for cleaning up our memory. there is no garbage collector for performance reasons, therefore when we are finished with allocated memory we must call a delete on the memory we allocated.

BACK TO TOP ^ **NEXT TUTORIAL >>**

App 09: using MPI_Scatter to distribute partitions of data to each node

In the previous example we had a partition of data that needed to be distributed among all nodes in the MPI cluster. In most cases if you have an array that needs to be distributed in partitions among different nodes. As this is a common operation in MPI there is a function dedicated to this operation. This function is called `MPI_Scatter`, and it works in a similar way to MPI_Bcast and MPI_Reduce.

01) start with a new MPI project

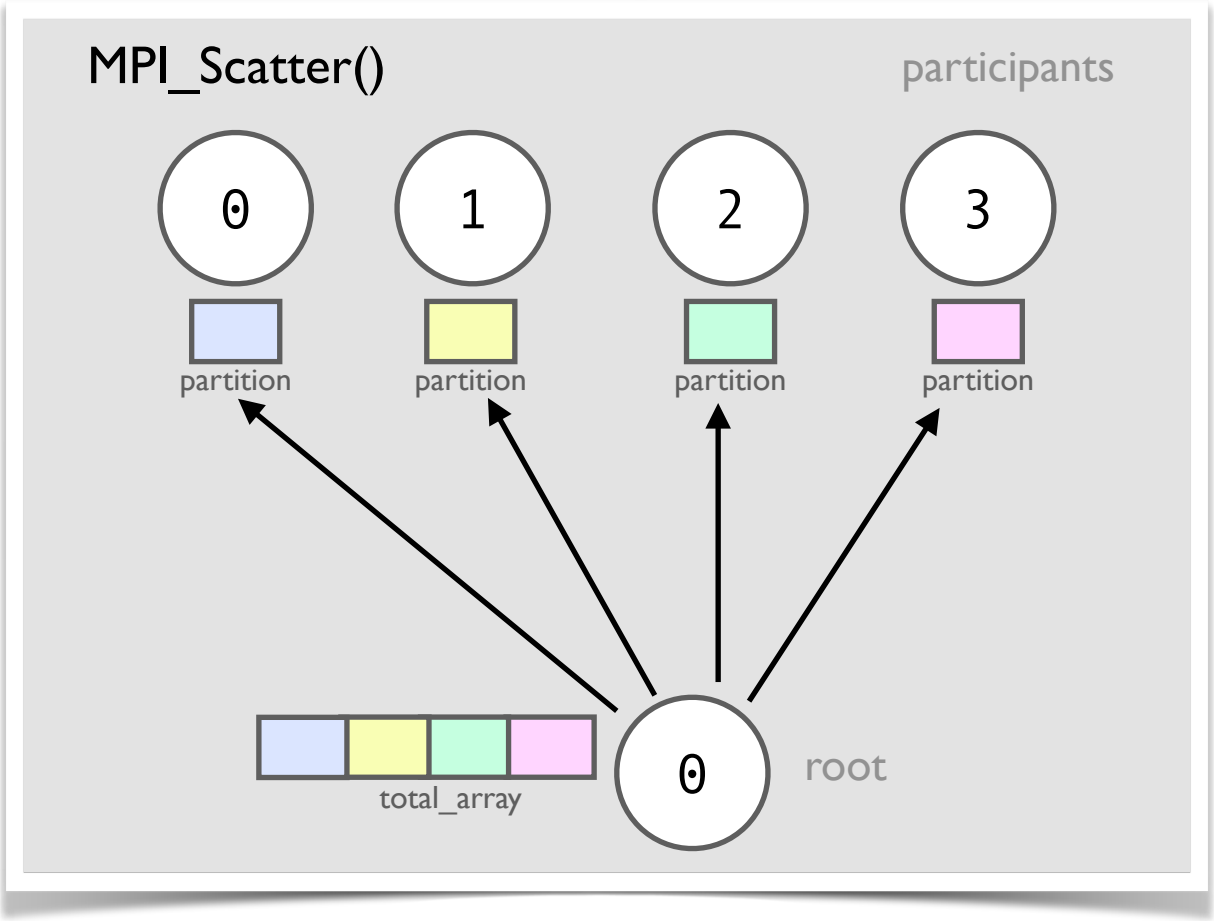
02) in your source file add the following code:

```
/** MPI_notes - App09 - using MPI_Scatter **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size and current rank
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // arrays that we need for communicating data
    int *total_array = new int[40];
    int *partition = new int[40 / world_size];

    // if we are process zero then create an array of 40 random integers
    if(world_rank == 0) {
        std::cout << "total array start-------- " << std::endl;
        for(unsigned int i = 0; i < 40; i++) {
            total_array[i] = rand() % 10;
            std::cout << total_array[i] << ", ";
            if(i % 10 == 9) std::cout << std::endl;
        }
        std::cout << "total array end-------- " << std::endl;
    }

    // run the scatter operation and then display the contents of all 4 nodes
    MPI_Scatter(total_array, 40 / world_size, MPI_INT, partition, 40 / world_size,
                MPI_INT, 0, MPI_COMM_WORLD);

    std::cout << "rank " << world_rank << " received partition: ";
    for(unsigned int i = 0; i < 40 / world_size; i++)
        std::cout << partition[i] << ", ";
    std::cout << std::endl;

    // always finalise the MPI library
    MPI_Finalize();

    // clear up our memory before we finish
    delete[] total_array;
    delete[] partition;
}
```

A few things to note here:

- In this example a 40 element array of random integers is created by rank 0 that is to be partitioned and distributed to all ranks in the communicator.
- We print out the array to begin with to show all of the values that are in the generated array to check that the scatter operation works as expected

- All ranks must participate in the MPI_Scatter command including the rank that is scattering the data to other nodes.
- The `MPI_Scatter` command expects 8 arguments:
  - The first argument is the array that is to be scattered amongst the ranks in the communicator.
  - The second argument is the number of items that should be distributed to each rank
  - The third is the type of data that is being distributed
  - the node that serves as the root will reference these three arguments while non-root nodes will reference the next three
  - The fourth, fifth and sixth arguments follow the same format as the first three but this time they describe the buffer that is receiving the data. again a location, count and data type that is being received
  - the last two arguments specify the root of the scatter and also which communicator that this scatter is functioning on.



BACK TO TOP ^

NEXT TUTORIAL >>

App 10: using MPI_Gather to receive partitions of data from all nodes

In the previous example we saw how to send individual partitions of data from a single root node to multiple child nodes. It is not uncommon for MPI applications to send data out to multiple nodes and expect to receive partitions of data back.

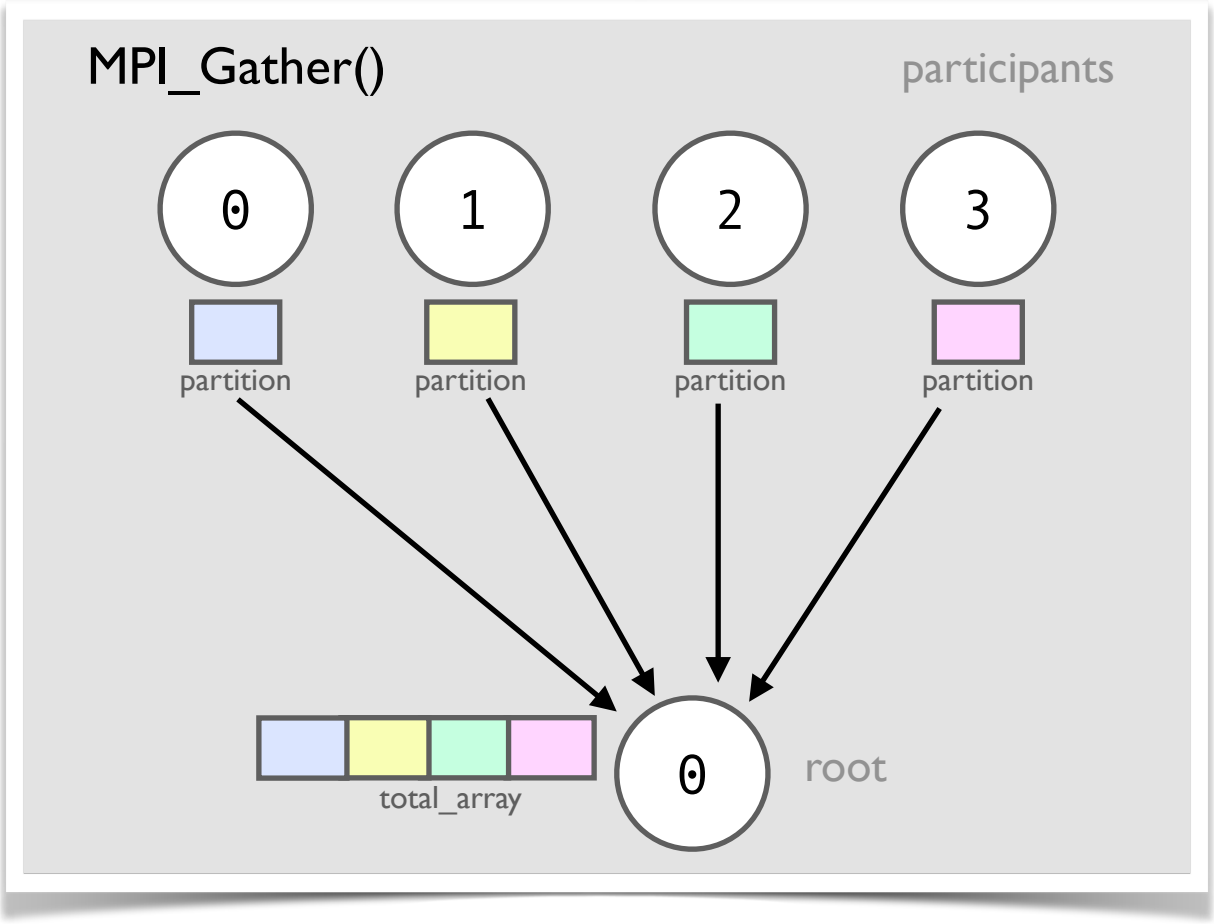
The example we show here is the opposite of App09 we have a set of nodes generating 10 random numbers and the root will gather all 4 partitions and will assemble them in a single array using `MPI_Gather`.

01) start with a new MPI project
02) give your source file the following shell code:

```
/** MPI_notes - App10 - using MPI_Gather **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size and current rank
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // arrays that we need for communicating data
    int *total_array = new int[40];
    int *partition = new int[40 / world_size];

    // all processes should generate a partition of data to be communicated to the
    // gathering node and print to console
    srand(world_rank);
    std::cout << "rank " << world_rank << " numbers: ";
    for(unsigned int i = 0; i < 40 / world_size; i++) {
        partition[i] = rand() % 10;
        std::cout << partition[i] << ", ";
    }
    std::cout << std::endl;

    // run the gather operation and print out the total partition that was recieved
    MPI_Gather(partition, 40/world_size, MPI_INT, total_array, 40/world_size, MPI_INT, 0,
        MPI_COMM_WORLD);

    if(world_rank == 0) {
        std::cout << "total data--------" << std::endl;
        for(unsigned int i = 0; i < 40; i++) {
            std::cout << total_array[i] << ", ";
            if(i % 10 == 9)
                std::cout << std::endl;
        }
        std::cout << std::endl;
    }

    //  always finalise the MPI Library
    MPI_Finalize();

    // clear up our memory before we finish
    delete[] total_array;
    delete[] partition;
}
```

A few things to note here:

- As stated this example is similar to the previous application but here we are *gathering* multiple partitions of data together
- All processes will generate their own individual partition of data before taking part in the gather operation
- Note that `MPI_Gather` takes the same number of arguments and the same types for each argument as MPI_Scatter
- Be careful! The size to send must match the size to receive as the root node is expecting this size from all nodes that are taking part in the gather.
  - If you specify the total size of the partition in the receive your MPI application will crash and will be shut down.



BACK TO TOP ^

NEXT TUTORIAL >>

App 11: creating a new communicator in MPI

In the examples thus far we have defaulted to using the single provided communicator `MPI_COMM_WORLD`. However for many MPI tasks it is useful to divide communications into separate subgroups. Each of these groups is a communicator but all communicators are subsets of `MPI_COMM_WORLD`. This example will show you how to make subsets of `MPI_COMM_WORLD`.

01) start with a new MPI project

02) give your source file the following shell code:

```
/** MPI_notes - App11 - Creating a new MPI Communicator **/

/** includes **/
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size and the world rank
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // code for 03 goes here

    // code for 04 goes here

    // code for 05 goes here

    // code for 06 goes here

    // always finalise MPI Library
    MPI_Finalize();
}
```

03) add in the following code in place of the comment marked "code for 03 goes here":

```
// creating a communicator in MPI requires a bit of work
// the 1st task is to extract a handle for the world communicator
// all user defined communicators are subsets of MPI_COMM_WORLD
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
```

A few things to note here:

- Building a communicator requires the use of MPI_Group.
- One of the first tasks you do in creating a new communicator is to retrieve the communicator associated with `MPI_COMM_WORLD` through the use of the MPI_Comm_group command.

04) add in the following code in place of the comment marked "code for 04 goes here":

```
// determine the world rank numbers for the new communicator.
// here we will divide our group into two dynamically
int *group_ranks = new int[world_size / 2];
if(world_rank < world_size / 2) {
    for(int i = 0; i < world_size / 2; i++)
        group_ranks[i] = i;
}
else {
    for(int i = world_size / 2, j = 0; i < world_size; i++, j++)
        group_ranks[j] = i;
}

// print out the group ranks
std::cout << "rank " << world_rank << " group ranks: ";
for(int i = 0; i < world_size / 2; i++)
    std::cout << group_ranks[i] << ", ";
std::cout << std::endl;
```

A few things to note here:

- Here we are dividing our world ranks into two separate groups. We have a group for the lower half of ranks (the if statement) and a group for the upper half of ranks (the else statement).
- The code you see here will work for any even number of processes that you provide

05) add in the following code in place of the comment marked "code for 05 goes here":

```
// create a new group out of the ranks and then create the communicator with that group
MPI_Group new_group;
MPI_Group_incl(world_group, world_size / 2, group_ranks, &new_group);
MPI_Comm sub_comm;
MPI_Comm_create(MPI_COMM_WORLD, new_group, &sub_comm);
```

A few things to note here:

- Using the ranks subsets that were created in 04 we can generate a new MPI_Group which in turn can be used to create a new communicator.
- `MPI_Group_incl` takes in a list of ranks to include in a new group and expects 4 arguments:
  - the first is the MPI_Group that you are making a subset of
  - the second is the number of ranks that will be in this new group
  - the third is the individual ranks that make up this group.
  - the fourth is the MPI_Group where this group is initialised and stored
- Once the new group is created we can initialise our new communicator by using the MPI_Comm_create function which expects three arguments
  - the first argument is the communicator that we are taking a subset from.
  - the second argument is the group of ranks that will form the subset
  - the third argument is the location of where to initialise and store the communicator.

06) add in the following code in place of the comment marked "code for 06 goes here"

```
// get our rank and size in the new communicator and print that out
int new_rank, new_size;
MPI_Comm_size(sub_comm, &new_size);
MPI_Comm_rank(sub_comm, &new_rank);
std::cout << "rank " << world_rank << " sub comm size: " << new_size << ", sub rank: "
<< new_rank << std::endl;

// send a simple broadcast message on the new communicator with the world rank
int message = world_rank;
MPI_Bcast(&message, 1, MPI_INT, 0, sub_comm);
std::cout << "world rank " << world_rank << " sub comm message is " << message << std::endl;

// delete all user defined communicators and groups
MPI_Comm_free(&sub_comm);
MPI_Group_free(&new_group);

// always deallocate memory when done
delete[] group_ranks;
```

A few things to note here:

- Note that communication on sub communicators work in the exact same way as using MPI_COMM_WORLD
- All communication functions that you have used so far will work with these new communicators
- Be aware that a process gets a new rank in each communicator it is part of. this rank is unique to this communicator.
- Note that at the end of your MPI application you are required to free your communicators and groups before finalising your application.

BACK TO TOP ^ **NEXT TUTORIAL >>**

App 12: using MPI_Scatterv to send different lengths of data to nodes

Sometimes in an MPI application you will need to send different sizes of data to different nodes. `MPI_Scatterv` is a variant of the normal scatter that will enable us to do this. However, the downside of this approach is that you need to setup an array with the offsets of each partition with a separate array detailing the length of each partition.

01) start with a fresh MPI project

02) give your source file the following shell code:

```
/** MPI_notes - App12 - using MPI_Scatterv **/

/** includes **/
#include <iostream>
#include <mpi.h>

// ** printArrayToConsole method goes here **

// ** coordinator method goes here **

// ** participant method goes here **

int main(int argc, char **argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size and world rank
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // decide which task this node is taking on depending on the rank
    if(world_rank == 0)
        coordinator(world_rank, world_size);
    else
        participant(world_rank, world_size);

    // always finalise the MPI library
    MPI_Finalize();
}
```

03) add in this function above the main function for printing out the data of an array. This will be used in some of the other functions later:

04) add in the following participant function just above the main function:

```
// function that takes in an integer array and prints it out to console
void printArrayToConsole(int *to_print, int print_size) {
    for(int i = 0; i < print_size; i++) {
        std::cout << to_print[i] << ", ";
        if(i % 10 == 9)
            std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

```
// task for the participant
void participant(int world_rank, int world_size) {
    // take part in the scatter to get our partition size
    int partition_size;
    MPI_Scatter(NULL, 0, MPI_INT, &partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    std::cout << "rank " << world_rank << " psize: " << partition_size << std::endl;

    // allocate memory for our partition
    int *partition = new int[partition_size];

    // take part in the scatterv to get our data
    MPI_Scatterv(NULL, NULL, NULL, MPI_INT, partition, partition_size, MPI_INT, 0, MPI_COMM_WORLD);

    // print out the numbers that we have obtained in the scatter
    printArrayToConsole(partition, partition_size);
```

```
    // delete our partition as we are finished with it
    delete[] partition;
}
```

A few things to note here:

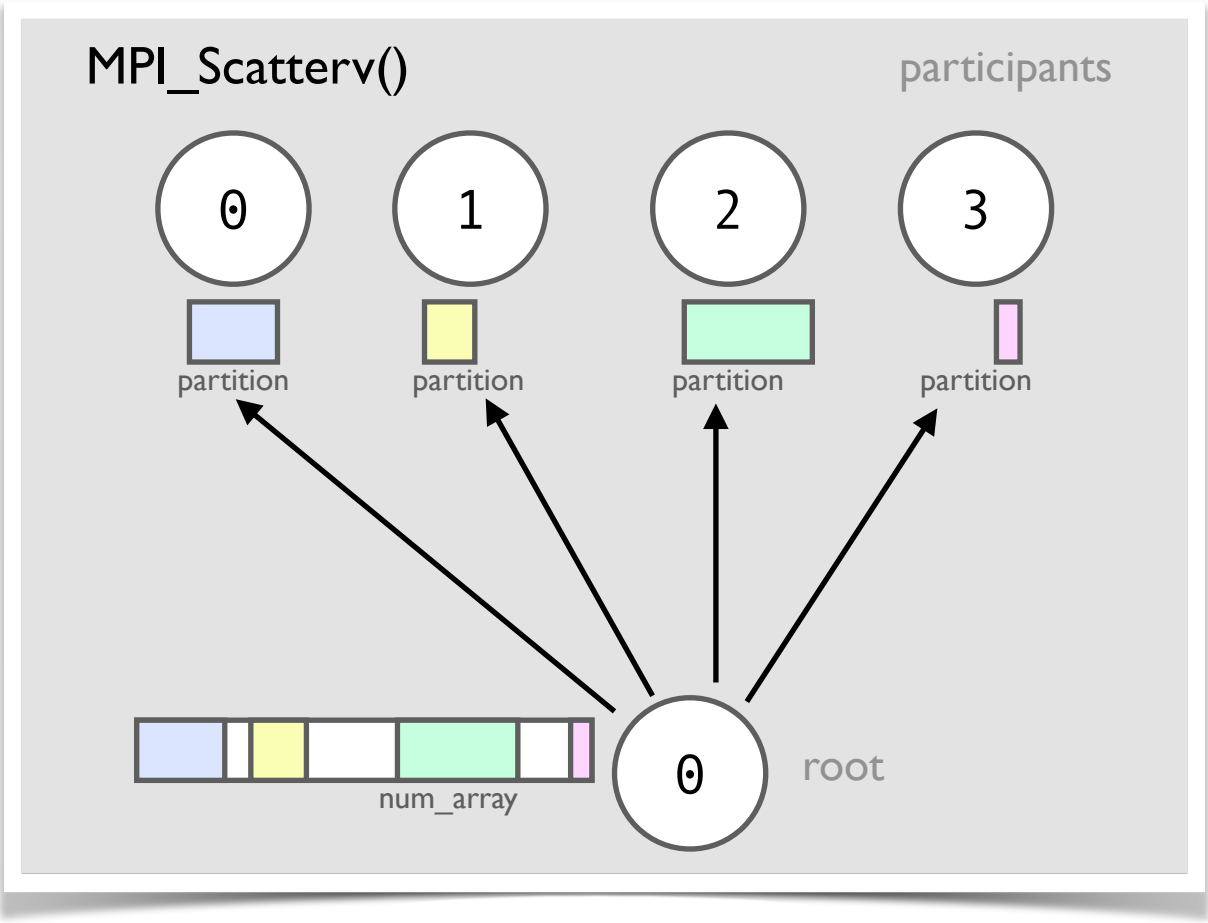
- To get the Scatterv to work, each node in the communicator should know what size of data it is receiving. This is why we have a normal scatter to begin as this is distributing the size of each partition to each node.
  - in this case we set the first two arguments to `NULL` and 0 because the receiver is not sending any data.
- Once each node knows the size of its partition we can take part in the `MPI_Scatterv`:
  - like the normal scatter the first 3 arguments are set to null because we are only receiving data. arguments 5 to 9 specify where to store the data, how much, what type who the sender is and what communicator this is working on
- Note that we are using the `Scatterv` function here. Normally a node that is a receiving in a scatter will set the first two arguments to NULL and 0 because the receiver is not sending data.

05) add in the following coordinator method above the participant function:

```
// task for the coordinator
void coordinator(int world_rank, int world_size) {
    // generate a group of 40 numbers randomly
    int *num_array = new int[40];
    for(unsigned int i = 0; i < 40; i++)
        num_array[i] = rand() % 10;
    printArrayToConsole(num_array, 40);

    // make a partition length array with even divisions, then randomly offset it by 1
    int *length_array = new int[world_size];
    for(unsigned int i = 0; i < 4; i++) {
        length_array[i] = (40 / world_size);
        if(i % 2)
            length_array[i]--;
        else
            length_array[i]++;
    }
    printArrayToConsole(length_array, 4);

    // make an offsets array by using the displacements array
    int *offsets_array = new int[world_size];
    offsets_array[0] = 0;
    for(unsigned int i = 1; i < 4; i++)
        offsets_array[i] = offsets_array[i-1] + length_array[i-1];
    printArrayToConsole(offsets_array, 4);

    // send each node their individual pariition size and allocate memory for our own partition
    int partition_size;
    MPI_Scatter(length_array, 1, MPI_INT, &partition_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    std::cout << "rank 0 psize: " << partition_size << std::endl;
    int *partition = new int[partition_size];

    // use the scatterv to send the data to each node
    MPI_Scatterv(num_array, length_array, offsets_array, MPI_INT, partition, partition_size,
        MPI_INT, 0, MPI_COMM_WORLD);

    // print out the numbers that we have obtained in the scatter
    printArrayToConsole(partition, partition_size);

    // delete our partition as we are finished
    delete[] partition;
    delete[] num_array;
    delete[] length_array;
    delete[] offsets_array;
}
```

A few things to note here:

- The first two loops set up random data and some uneven partition lengths
  - the partition lengths determine how much data to send to each node we must have a length for each and every node
- The following loop sets up offsets in the data to determine where in the array to start sending data to each node.
  - this combined with the lengths array will determine what each and every node will receive in terms of data
  - for example if we have lengths of 11,9,11,9 and offsets of 0,11,20,31 this will send 11 items to rank zero starting from offset 0. rank 1 will get 9 items of data starting from offset 11 etc etc
- The first of the scatters sends the length of each partition to the appropriate node in the communicator so they can set up their partitions correctly and expect the right size of data
- The Scatterv is next. The first three arguments are the most important for the root these specify the data that is to be sent to all nodes, the length of data to be sent to each node (must be an array) and the offsets of each partition in the data (must be in an array).



BACK TO TOP ^

NEXT TUTORIAL >>

App 13: using an MPI_Gatherv to get different sizes of data

In this example we will show a similar action to App12. Instead of scattering different sizes of data we will gather in different sizes of data. The arguments to the `MPI_Gatherv` function are similar to Scatterv except this time all nodes are sending and only the root is receiving.

01) start with a new MPI project

02) give your source file the following shell code:

```
/** MPI_notes - App13 - Using MPI_Gatherv **/

/** includes **/
#include <iostream>
#include <mpi.h>

// ** printArrayToConsole method goes here **

// ** coordinator method goes here **

// ** participant method goes here **

int main(int argc, char **argv) {
    // initialise the MPI library
    MPI_Init(NULL, NULL);

    // determine the world size and world rank
    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // decide which task this node is taking on depending on the rank
    if(world_rank == 0)
        coordinator(world_rank, world_size);
    else
        participant(world_rank, world_size);

    // we are done with the MPI library so we must finalise it
    MPI_Finalize();
}
```

03) add in the following function above the main function:

```
// takes in an integer array & prints it out to console
void printArrayToConsole(int *to_print, int print_size) {
    for(int i = 0; i < print_size; i++) {
    std::cout << to_print[i] << ", ";
        //if(i % 10 == 9)
            // std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

04) add in the following participant function:

```
// task for the participant
void participant(int world_rank, int world_size) {
    // generate some random data
    int partition_size = (40 / world_size);
    if(world_rank % 2 == 0)
        partition_size++;
    else
        partition_size--;

    int *partition = new int[partition_size];
    srand(world_rank);

    for(unsigned int i = 0; i < partition_size; i++)
        partition[i] = rand() % 10;

    std::cout << "rank " << world_rank << " data";
    printArrayToConsole(partition, partition_size);
```

```
    // take part in gather to let root know how much data to expect
    // participant doesn't need to send any data so args 4 & 5 are null & zero
    MPI_Gather(&partition_size, 1, MPI_INT, NULL, 0, MPI_INT, 0, MPI_COMM_WORLD);

    // take part in gatherv to send all our data to the root
    MPI_Gatherv(partition, partition_size, MPI_INT, NULL, NULL, NULL, MPI_INT, 0, MPI_COMM_WORLD);

    // clean up allocated data
    delete[] partition;
}
```

A few things to note here:

- Each node generates a different partition of data of a different size
- Before the master can gather all of this data it must first find out how much data each node is sending. this is why we use the first normal gather
- The `Gatherv` afterwards sets the data that each node will send to the root (defined by the first four arguments)
  - Like before with the `Scatterv` the participant is not receiving data which is why the next three arguments are set to null.

05) add in the following coordinator function

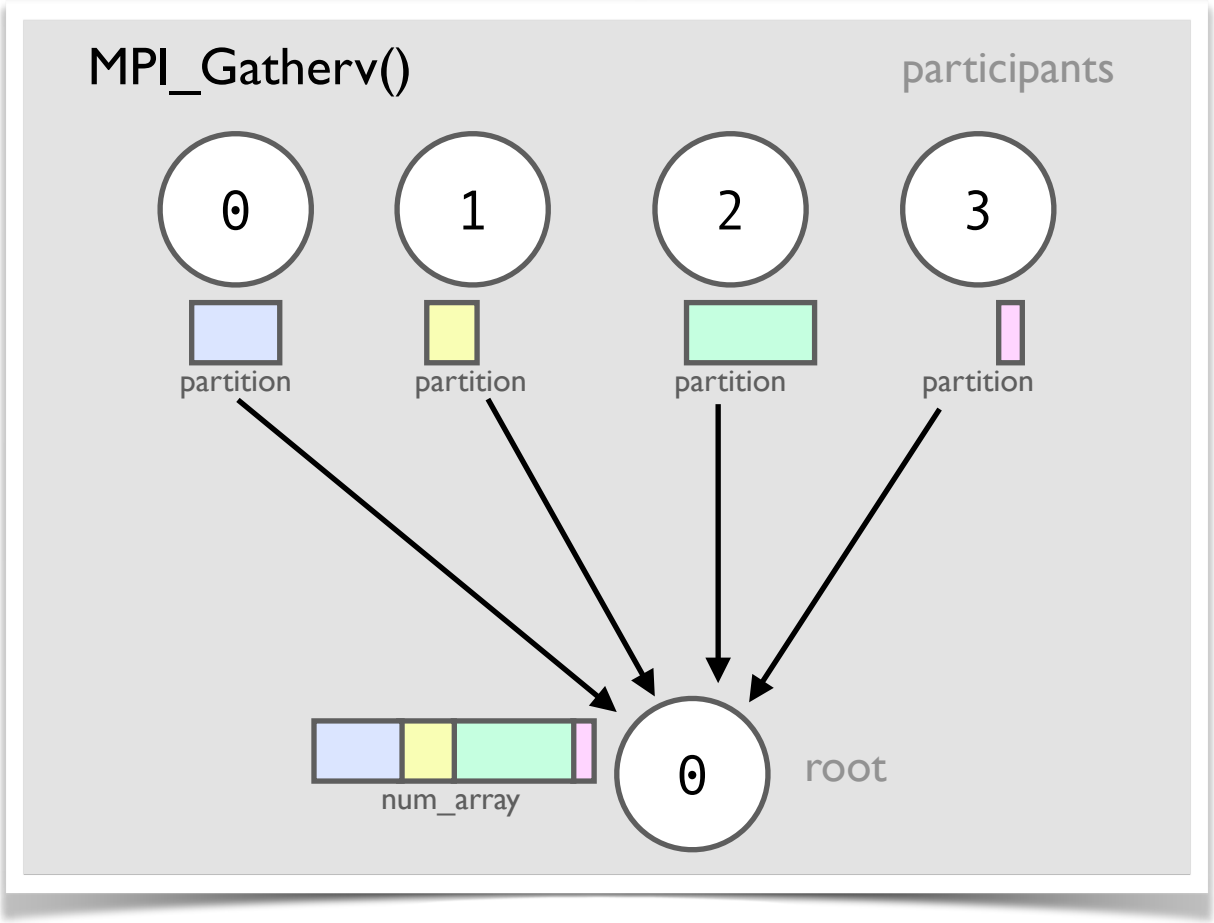
```
// task for the coordinator
void coordinator(int world_rank, int world_size) {
    // generate some random data as we are node zero
    int partition_size = (40 / world_size) + 1;
    int *partition = new int[partition_size];
    int *num_array = new int[40];

    for(unsigned int i = 0; i < partition_size; i++)
        partition[i] = rand() % 10;

    std::cout << "rank 0 data";
    printArrayToConsole(partition, partition_size);

    // gather the length of each partition that each node has
    int *lengths_array = new int[world_size];

    // generate the offsets for each partition before we take part in the gatherv
    MPI_Gather(&partition_size, 1, MPI_INT, lengths_array, 1, MPI_INT, 0, MPI_COMM_WORLD);

    std::cout << "rank 0 sizes recieved: ";
    printArrayToConsole(lengths_array, world_size);

    int *offsets_array = new int[world_size];
    offsets_array[0] = 0;

    for(unsigned int i = 1; i < 4; i++)
        offsets_array[i] = offsets_array[i-1] + lengths_array[i-1];

    // gather the data from the nodes
    MPI_Gatherv(partition, partition_size, MPI_INT, num_array, lengths_array, offsets_array,
        MPI_INT, 0, MPI_COMM_WORLD);

    // display the data
    std::cout << "rank 0 recieved data: " << std::endl;
    printArrayToConsole(num_array, 40);

    // clean up any memory we have allocated
    delete[] lengths_array;
    delete[] partition;
    delete[] num_array;
}
```

A few things to note here:

- Similar to App12 the root needs to figure out how much data each node is going to send, which is why we need the first gather.

- From this data we need to determine the offsets for where to store the data in our array.

- The `Gatherv` that follows defines what data the root is going to send to itself (first 4 arguments) and where to store all gathered data next 3 arguments
  - the 3 arguments must all be array types that specify the storage area, the amount data to be received from each node and where to store each set of data in the array itself.



BACK TO TOP ^